VI.08.

Общий обзор.

Когда мы начинали говорить про интерфейсы, мы рассматривали всякие примеры, связанные с сортировками: сортированные и сортируемые коллекции и списки, бинарная куча, бинарное дерево поиска, АВЛ-дерево. Во всех этих примерах мы делали данные поумнее - приводили их к интерфейсу `Ordered`:

```
type Ordered interface {
    Before(b Ordered) bool
    Show() string
}
```

Главное в этом интерфейсе, конечно, - метод `Before`, который позволяет любому элементу сравниваться с другими элементами. И тут мы наталкиваемся на то, что хотя мы строим какие-то структуры данных, но требуем мы от данных достаточно много, чтобы они были довольно умными - умели сравниваться. И это расходится с идеологией Go. Данные должны быть как можно натуральнее, не отягощенные особым интеллектом. А где тогда место для разума? А в структуре - ведь расстановка данных есть функция структуры, а не данных, данные сами по себе не сортируются, не взаимодействуют с другими данными, они начинают как-то взаимодействовать только в рамках структуры, в рамках какого-то объединения. И попробуем теперь взглянуть на все структуры, что мы строили (а строили мы их, разбираясь с интерфейсами) несколько с иной позиции, более Go-шной, если можно так выразиться. И этим мы и займёмся в ближайшие занятия.

К вопросу о подходе и интеллектуализации данных, можно говорить ещё и о том, что обработка таких вот, не совсем натуральных (грубо говоря, больно умных), данных требует их преобразования перед обработкой, что это требует определённых ресурсов - и памяти, и времени процессора, но это так, мелочи, это не есть принципиальный вопрос, экономия ресурса - это, скорее, побочный эффект Go-подхода, приятный, но всего лишь бонус. Главное - это способ осознания задачи, данных, их места в процессе обработки.

А сегодня мы для начала посмотрим на то, как это делают авторы Go: посмотрим, как в стандартной библиотеке реализуются простые связные структуры данных: двусвязный список и двусвязное кольцо. А дальше заглянем в `package sort` и увидим, что сортируемая коллекция (то, что мы называли `SortableCollection`) там уже реализована, посмотрим как

именно. А в `package container/heap` реализована бинарная куча - тоже обязательно посмотрим. А вот сортированную колллекцию (`SortedCollection`), которая всегда отсортирована, а не сортируется по команде, реализуем самостоятельно, но уже с новых позиций. Дальше, конечно, надо говорить про сортированные/сортируемые списки, но тут уж как успеем. По ходу пьесы будем разглядывать стандартные библиотеки, которые успешно можно найти в `$GOROOT$/src` , но на всякий случай и для полноты материала все рассматриваемые пакеты продублируем прямо в материалах занятия в каталоге `./packages` .

Чтение исходников - это не самый простой навык, но он очень важен, он просто жизненно необходим, так что у сегодняшнего занятия есть ещё и дополнительная существенная функция - учимся читать исходники. И это очень существенная часть **задания для самостоятельной работы**, это важно произнести явно и неоднократно. Ну, а чтение исходников от достойных авторов - это не только труд, это ещё и удовольствие, тем большее, что добывается оно в трудах.

package container/list. Двусвязный список. Файл list.go

Первым делом посмотрим на объявленные типы:
тип элемента списка

```
// Element is an element of a linked list.
type Element struct {
    // Next and previous pointers in the doubly-linked list of elements.
    // To simplify the implementation, internally a list l is implemented
    // as a ring, such that &l.root is both the next element of the last
    // list element (l.Back()) and the previous element of the first list
    // element (l.Front()).
    next, prev *Element

    // The list to which this element belongs.
    list *List

    // The value stored with this element.
    Value interface{}
}
```

и тип списка

```
// List represents a doubly linked list.
// The zero value for List is an empty list ready to use.
type List struct {
    root Element // sentinel list element, only &root, root.prev, and root.next are
used
    len  int     // current list length excluding (this) sentinel element
}
```

Что мы сразу видим? Первое, на что падает взгляд - это то, что значение (`Value`) собственно связываемых данных имеет тип пустого интерфейса (`interface {}`). Мы, конечно, знаем, что пустой интерфейс - штука опасная, хоть в неё влазит всё, но при получении пустого интерфейса полученное значение не говорит само по себе о себе ничего, т.е. мы обязаны будем ассертить получаемый пустой интерфейс. Но тут уж ничего не поделаешь - библиотека в самом деле предначначена для хранения величин произвольных типов. И, может быть тут не совсем место для следующего пассажа, но давайте прямо сейчас обговорим этот момент подробно, дальше будем двигаться побыстрее. Почему же мы в начале сказали, что использовать для структур с сортировкой интерфейс `Ordered` не есть хорошая политика? А дело вот в чём. Ну, что с того, что мы будем получать из структуры не пустой интерфейс, а величину интерфейсного типа `Ordered` ? Ведь этот интерфейс даёт нам исключительно свойства, необходимые для вставки их в структуру, для взаимодействия элементов структуры. Получается, что вставляя в структуру величины типа `Ordeerd` , мы заметно усложняем себе работу, не получая ны выходе вообще ничего - когда мы получаем значение, нам по барабану, как там его сравнивали с другими величинами, нас интересует само значение, а не его отношения с другими величинами. Для выяснения отношений умы и создавали структуру, и вставляли элементы в неё, организовывали их взаимоотношения. Так и ну его нафиг этот `Ordered` , что он нам…

Это был принципиальный вопрос. Но поехали дальше…

Простенький пример listest.go

```
package main

import (
    "container/list"
    "fmt"
)
```

```
func main() {
    var l list.List
    fmt.Printf("%v\n", l) // {{<nil> <nil> <nil> <nil>} 0}
    l.Init()
    fmt.Printf("%v\n", l) // {{0xc000074480 0xc000074480 <nil> <nil>} 0}
    ll := list.New()
    fmt.Printf("%v\n", *ll) // {{0xc000074510 0xc000074510 <nil> <nil>} 0}
    e := l.PushFront(777)
    fmt.Printf("%v\n", l)  // {{0xc000074570 0xc000074570 <nil> <nil>} 1}
    fmt.Printf("%v\n", *e) // {0xc000074480 0xc000074480 0xc000074480 777}
    l.PushFront(1234)
    fmt.Println(*l.Front(), *l.Back())
    // {0xc000074570 0xc000074480 0xc000074480 1234} {0xc000074480 0xc000074600 0xc0
00074480 777}
}
```

полностью подтверждает то, что написано в описании типов `List` и `Element`, а также функциq `New(), Front(), Back()`, что, впрочем, совершенно естественно. Хорошо бы проиллюстрировать список в процессе его жизни в примере рисунками. Но они совсем простые, так что не будем здесь их рисовать - больно геморройно это делать в текстовом формате. Обратим внимание на то, что оба пОля в `List` и все полЯ, кроме пОля `Value`, в `Elenent` приватные (неэкспортируемые), так что доступаться к ним можно только через методы пакета. И это тоже совершенно естественно и правильно, но на это стОит явно обратить внимание. И также естественно, раз уж мы решили держать список и за хвост, и за голову, использовать тип узла для хранения всего списка. Разве что у этого узла не должно быть `Value`. Оно, конечно, есть, но обратиться к нему снаружи совершенно невозможно, а изнутри пакета обращений к нему нет - хорошо бы в этом убедиться. Правда ещё в чисто реализационных интересах в списке добалено поле для длины списка, и оно тоже не экспорируется, т.е. внешний доступ к нему обеспечивают методы, точнее один метод Len(), причём только для чтения. Чудно всё срастается.

Ещё интересно посмотреть на методы с точки зрения их экспортируемости/неэкспортируемости (неэкспортируемые методы, как и всё остальное, начинаются с маленькой буквы, точнее их названия начинаются с маленькой буквы - так, напомним на всякий случай). Вот, например, метод `lazyInit`. Используется он только в методах вставки чего-то в список - элемента или другогол списка: `PushFront, PushBack, PushFrontList, PushBackList` - для того, чтобы не пытаться вставить в nil-список, чтобы сначала исходны список уже существовал, хотя бы и как пусто список, а пустой список - это список длины 0, а

не nil-список, сентинел (сторож, часовой, если дословно) `root` в списке должен быть. Понятно, что метод `lazyInit` чисто имплементационный, что экспортировать его совершенно ни к чему. Аналогично обстоят дела и с прочими неэкспортируемыми методами - `insert, insertValue, remove, move` - но хорошо бы посмотреть на их предназначение и использование. По ходу естественным образом становятся понятны и реализации экспортируемых методов `Remove, PushFront, PushBack, InsertBefore, InsertAfter, MoveToFront, MoveToBack, MoveBefore, MoveAfter, PushBackList, PushFrontList` .

package container/ring. Кольцо. Файл ring.go

В принципе, тут всё очень похоже, так что особо разливаться тут не станем - больше для самостоятельной работы это пакет. Но несколько моментов всё-таки обговорим.

Во-первых, сентинела у кольца нет. Резонно - ведь в кольце все элементы равноправны ("…а у кольца начала нет и нет конца…"), мы держим кольцо за любой элемент, разницы между ними нет. Соответственно, для хранения длины кольца места не предусмотрено, так что функция `Len()` работает небыстро, пробегая по всему кольцу, чтобы определить его длину. Любопытно, но очень естественно, нет специального типа для элемента кольца - да, каждый элемент кольца есть кольцо и ссылается на кольцо. Добавление элемента в кольцо отражает эту мысль - для этого надо создать кольцо из одного элемента, а затем вклеить это кольцо в исходное кольцо с помощью метода `Link` . Эту функцию, как и функцию `New(n int) *Ring` , очень похожую по реализации, есть смысл посмотреть самостоятельно - там всё просто и естественно. СтОит отметить, что только так и надо вставлять новые элементы или создавать новые кольца - такой подход гарантирует, что у нас никакой элемент не окажется в кольце дважды, новые элементы создаются функциями пакета, а вот за начинку элементов (за `Value`) отвечает пользователь. А повторное появление элемента в кольце - это ужоасно, ведь пакет определяет конец кольца при его обходе именно по появлению в обходе начального элемента обхода. Так делают методы `Do` и `Len` . Интересно также разобраться с методом `Unlink` .

Для самостоятельной работы можно дать пару таких заданий:

Односвязное кольцо.
Просто реализовать однонаправленное кольцо - каждый элемент смотрит только на следующий элемент кольца.

```
type Ring struct {
    next *Ring
    Value      interface{} // for use by client; untouched by this library
```

```
}
```

Разумеется, здесь не будет метода `Prev()`, а метод `Move(n int)` должен игнорировать отрицательные значения параметра.

Односвязная очередь.
Реализовать очередь с помощью односвязного списка. По аналогии с `package list` объявим типы очереди

```
type Queue struct {
    head *Element
    tail *Element
    len  int
}
```

и элемента очереди

```
type Element struct {
    next *Element

    // The queue to which this element belongs.
    queue *Queue

    // The value stored with this element.
    Value interface{}
}
```

Требуется реализовать вот такие методы:

```
// New returns an initialized queue.
// head, tail, len = nil, nil, 0
// Собственно, этп реализация уже готова
func New() *Queue { return new(Queue) }

// Len returns the number of elements of queue q.
// The complexity is O(1).
func (q *Queue) Len() int { return q.len }

// Head returns the first element of queue q or nil if the queue is empty.
func (q *Queue) Head() *Element {
```

```
// Tail returns the last element of queue q or nil if the queue is empty.
func (q *Queue) Tail() *Element {

// Push inserts a new element e with value v at the tail of queue q and returns e.
func (q *Queue) Push(v interface{}) *Element {

// Pop returns the head element e and removes that from queue.
func (q *Queue) Pop() *Element {
```

Тут можно легко продолжить список таких заданий, но они все на повторение, они не очень привязаны к нашим текущим штудиям.

package sort. Взгляд по поверхности.

Почему только по поверхности? Дело в том, что подавляюще бОльшая часть пакета описывает и реализует несколько методов сортировки, которые весьма нетривиально сочетаются в главном экспортируемом из пакета методе `Sort`. Собственно, кроме этого метода пакет `sort` экспортирует только сортировку в обратном порядке (по убыванию, грубо говоря) и специализации сортировки слайсов стандартных типов: `int`, `float64` и `string`. Отдельная история про устойчивую сортировку, которую также предоставляет `package sort`. Но это другая история. О ней точно не сегодня. А для нас интересен сегодня определённый в пакете интерфейс `Interface` (с большой буквы, да). О нём и поговорим сейчас, перемежая разговоры цитатами из текста пакета `sort` (цитаты в более или менее точном переводе выделяются как go-тексты)

```
//Пакет sort предоставляет примитивы для сортировки слайсов и пользовательских колле
кций.
// Реализация интерфейса Interface может быть отсортирована, пользуясь подпрограммам
и в этом пакете.
//Методы ссылаются на элементы базовой коллекции по целочисленному индексу.
```

Грубо говоря `Interface` - это интерфейс, обобщающий сортируемые слайсы или другие коллекции, обеспечивающие индексный доступ к своим элементам. На них и ориентирован пакет `sort`.

```
type Interface interface {
    // Len - это количество элементов в коллекции.
    Len() int
```

```
    // Less сообщает, должен ли элемент с индексом i располагаться
    // в результате сортировка перед элементом с индексом j.
    //
    // Если и Less(i, j), и Less(j, i) ложны, то элементы с
    // индексами i и j считаются равными.
    //
    // func Sort(data Interface) может размещать в конечном
    // результате равные элементы в любом порядке, в то время как
    // func Stable(data Interface) сохраняет исходный порядок
    // равных элементов.
    //
    // Less должно обладать свойствами транзитивности:
    //  - если и Less(i, j), и Less(j, k) истинны, то Less(i, k) также должно быть и
стинным.
    //  - если и Less(i, j), и Less(j, k) являются ложными, то Less(i, k) также долж
но быть ложным.
    //
    Less(i, j int) bool

    //Swap меняет местами элементы с индексами i и j.
    Swap(i, j int)
}
```

Может быть, есть смысл посмотреть на, простую в общем-то, штуку - на специализации сортировки слайсов стандартных типов: `int`, `float64` и `string`. Вся их суть - дать пользователю не ассертить получаемые результаты, если они являются `int`ами, `float64`ами или `string`ами. Для этого, во-первых, реализуется интерфейс `Interface` для всех таких видов слайсов. Особое внимание стОит обратить на реализацию метода `func (x Float64Slice) Less(i, j int) bool`, который отдельно учитывает значение `NaN`. Во-вторых, для каждого из этиз трёх видов слайсов определяется метод `Sort`, который просто вызывает для них функцию `Sort`. Ну, и, в-третьих, создаются лёгенькие (а они практически всегда лёгенькие) обёртки (врапперы, wrappers) для сортировки слайса и проверки, отсортирован ли слайс - это для того, чтобы пользователь мог даже не заморачиваться на фактически чисто внутренние, хотя и экспортируемые, типы `IntSlise`, `Float64Slice`, `StringSlice` а просто пользоваться слайсами `[]int`, `[]float64`, `[]string`. Вообще, обёртки/врапперы - это довольно естественая и распространённая вещь, но об этом в другой раз…

```
// Convenience types for common cases
```

```
// IntSlice attaches the methods of Interface to []int, sorting in increasing order.
type IntSlice []int

func (x IntSlice) Len() int           { return len(x) }
func (x IntSlice) Less(i, j int) bool { return x[i] < x[j] }
func (x IntSlice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

// Sort is a convenience method: x.Sort() calls Sort(x).
func (x IntSlice) Sort() { Sort(x) }

// Float64Slice implements Interface for a []float64, sorting in increasing order,
// with not-a-number (NaN) values ordered before other values.
type Float64Slice []float64

func (x Float64Slice) Len() int { return len(x) }

// Less reports whether x[i] should be ordered before x[j], as required by the sort
Interface.
// Note that floating-point comparison by itself is not a transitive relation: it do
es not
// report a consistent ordering for not-a-number (NaN) values.
// This implementation of Less places NaN values before any others, by using:
//
//    x[i] < x[j] || (math.IsNaN(x[i]) && !math.IsNaN(x[j]))
//
func (x Float64Slice) Less(i, j int) bool { return x[i] < x[j] || (isNaN(x[i]) && !i
sNaN(x[j])) }
func (x Float64Slice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

// isNaN is a copy of math.IsNaN to avoid a dependency on the math package.
func isNaN(f float64) bool {
    return f != f
}

// Sort is a convenience method: x.Sort() calls Sort(x).
func (x Float64Slice) Sort() { Sort(x) }

// StringSlice attaches the methods of Interface to []string, sorting in increasing
order.
type StringSlice []string
```

```
func (x StringSlice) Len() int           { return len(x) }
func (x StringSlice) Less(i, j int) bool { return x[i] < x[j] }
func (x StringSlice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

// Sort is a convenience method: x.Sort() calls Sort(x).
func (x StringSlice) Sort() { Sort(x) }

// Convenience wrappers for common cases

// Ints sorts a slice of ints in increasing order.
func Ints(x []int) { Sort(IntSlice(x)) }

// Float64s sorts a slice of float64s in increasing order.
// Not-a-number (NaN) values are ordered before other values.
func Float64s(x []float64) { Sort(Float64Slice(x)) }

// Strings sorts a slice of strings in increasing order.
func Strings(x []string) { Sort(StringSlice(x)) }

// IntsAreSorted reports whether the slice x is sorted in increasing order.
func IntsAreSorted(x []int) bool { return IsSorted(IntSlice(x)) }

// Float64sAreSorted reports whether the slice x is sorted in increasing order,
// with not-a-number (NaN) values before any other values.
func Float64sAreSorted(x []float64) bool { return IsSorted(Float64Slice(x)) }

// StringsAreSorted reports whether the slice x is sorted in increasing order.
func StringsAreSorted(x []string) bool { return IsSorted(StringSlice(x)) }
```

package heap.

А вот тут поподробнее. На всякий случай, поместим для напоминания материал про бинарную кучу.

И опять включим цитаты из `package heap`.

```
// Package heap provides heap operations for any type that implements
// heap.Interface. A heap is a tree with the property that each node is the
// minimum-valued node in its subtree.
```

```
//
// The minimum element in the tree is the root, at index 0.
//
// A heap is a common way to implement a priority queue. To build a priority
// queue, implement the Heap interface with the (negative) priority as the
// ordering for the Less method, so Push adds items while Pop removes the
// highest-priority item from the queue.
//
package heap

import "sort"

type Interface interface {
    sort.Interface
    Push(x interface{}) // add x as element Len()
    Pop() interface{}   // remove and return element Len() - 1.
}

// The Interface type describes the requirements
// for a type using the routines in this package.
// Any type that implements it may be used as a
// min-heap with the following invariants (established after
// Init has been called or if the data is empty or sorted):
//
//    !h.Less(j, i) for 0 <= i < h.Len() and 2*i+1 <= j <= 2*i+2 and j < h.Len()
//
// Note that Push and Pop in this interface are for package heap's
// implementation to call. To add and remove things from the heap,
// use heap.Push and heap.Pop.
```

Да, понятно, что, раз уж в бинарной куче идёт речь о сравнениях и перестановках элементов слайса, иначе говоря, бинарная куча - это тоже какая-то сортировка данных слайса (ну, или другой структуры с индексным доступом), специфическая, неоднозначная, но сортировка, так бинарная куча должна включать все методы интерфейса `sort.Interface`, его и встраиваем в `heap.Interface`, который и есть интерфейс бинарной кучи. Заметим, что бинарная куча просто по построению - есть индексированная структура, так что использовать `sort.Interface` совершенно естественно. Ну, а основное предназначение бинарной кучи - это иметь возможность динамически получать наименьший из имеющихся элементов (корневой), причём не только получать, но и извлекать/изымать его из кучи, так что нужен метод для изъятия элемента из кучи. Динамически - это означает,

что элементы могут подходить/добавляться в кучу по ходу пьесы, так что нужен и метод добавления элемента в кучу. Это и есть методы `Push(x interface{})` и `Pop() interface{}`.

Реализация методов пакета очень прозрачна и не требует особых комментариев. Разве что надо бы приостановиться и отчётливо разделить методы `Push` и `Pop` интерфейса `heap.Interface` от функций пакета с такими же названиями.

Пара примеров на кучу приводится. Первый из них - пример intheaptest.go - это тестовый пример `example_intheap_test.go` от авторов Go с минимальными изменениями. Изменения вызваны тем, что в оригинале этот файл имеет формат тестового пакета. О тестировании разговор отдельный, так что сейчас мы назовём пакет `main` и тестовую функцию тоже назовём `main`. Второй пример - пример workerheaptest.go - показывает как использовать бинарную кучу для сортировки нашего старого знакомого - слайса с работниками.

Но вернёмся к нашим структурам. Сегодня, пожалуй, успеем поговорить о сортированных/сортируемых коллекциях/списках: `Sorted/Sortable Collection/List`.

Говорить о `SortableCollection` уже поздно - `package sort` уже решил все наши проблемы. А вот `SortedColection` есть смысл рассмотреть, не говоря уже о списках.

SortedCollection

Это, напомню, слайс (ну, или произвольная структура с индексным доступом к элементам), который всегда отсортирован. Т.е. добавлять новые элементы можно только по одному и с помощью специального метода, скажем `Add`. Раз это слайс с некоторым упорядочением, то есть смысл сделать его интерфейсом `sort.Interface`. Так что всё, что надо сделать для добавления элемента - это:

```
    collection = append(collection, value)
    collection = Add(collection)
```

Фуекция `Add` последний элемент слайса переносит на присущее ему место. Считаем, что все элементы, кроме последнего, уже отсортированы. Тут всё просто, так что оставим это **в качестве задания для самостоятельной работы**. Задание простое, но оно поможет понять смысл всех движений и рычагов пакета `sort`.

SortedList.

Будем рассматривать односвязный список - с ним как-то поинтереснее поиграться. Но и про двусвязный не забудем.

В принципе, как делают список, пусть и двусвязный, авторы Go, мы видели. Вот и будем ориентироваться на хорошие примеры. Понятно, что нам нужно объявить тип элемента списка и тип списка. Список мы строим односвязный, значит каждый элемент смотрит только вперёд. Ну, и нормально. Но мы ведь строим список с сортировкой, т.е. надо как-то элементы упорядочивать-сравнивать. В начале семестра мы доверяли эту функцию элементам, но тут вспомним, что говорилось в **Общем обзоре занятия**: *И тут мы наталкиваемся на то, что хотя мы строим какие-то структуры данных, но требуем мы от данных достаточно много, чтобы они были довольно умными - умели сравниваться. И это расходится с идеологией Go. Данные должны быть как можно натуральнее, не отягощенные особым интеллектом. А где тогда место для разума? А в структуре - ведь расстановка данных есть функция структуры, а не данных, данные сами по себе не сортируются, не взаимодействуют с другими данными, они начинают как-то взаимодействовать только в рамках структуры, в рамках какого-то объединения. И попробуем теперь взглянуть на все структуры, что мы строили (а строили мы их, разбираясь с интерфейсами) несколько с иной позиции, более Go-шной, если можно так выразиться.* Ну, так давайте и делегируем функцию сравнивания элементов списка именно списку. Т.е. сделаем функцию сравнения элементов частью списка. Да, именно так - добавим в список ещё одно поле:

```
    less func(*Lmnt, *Lmnt) bool
```

И задавать его будет пользователь списка. Да, конечно, надо ещё научить список использовать функцию сравнения для вставки элемента - это делает метод `Add`. В общем, проще посмотреть на код программы sortedlist1.go:

```
package main

import (
    "fmt"
)

type Lmnt struct {
    Next  *Lmnt
    Value int
}

type SortedList struct {
    Head Lmnt // sentinel list Lmnt;
    // it's located at the top of the list
```

```
    Len  int // current list length excluding sentinel
    less func(*Lmnt, *Lmnt) bool
}

func NewSortedList(less func(*Lmnt, *Lmnt) bool) SortedList {
    return SortedList{less: less}
}

func (l *SortedList) Add(x int) {
    p := &(l.Head)
    v := &Lmnt{Value: x}
    for i := 0; i < l.Len; i++ {
        if l.less(v, (*p).Next) {
            break
        }
        p = (*p).Next
    }
    (*v).Next = (*p).Next
    p.Next = v
    l.Len++
}

func main() {
    before := func(p, q *Lmnt) bool {
        return (*p).Value > (*q).Value
    }
    l := NewSortedList(before)
    l.Add(2)
    l.Add(4)
    l.Add(1)
    l.Add(8)
    l.Add(5)
    p := l.Head
    for i := 0; i < l.Len; i++ {
        p = *(p.Next)
        fmt.Printf("%v\n", p.Value)
    }
}
```

Ну, а дальше совершенно естественное движение - убрать описание списка (и элемента списка) в отдельный `package`. Так и сделаем.

package sorted, который помещаем в `$GOROOT$/src/order/sorted` :

```
package sorted

type Element struct {
    Next *Element
    Value int
}

type SortedList struct {
    Head Element // sentinel list Element;
                 // it's located at the top of the list
    Len  int  // current list length excluding sentinel
    less func(*Element, *Element) bool
}

func NewSortedList (less func(*Element, *Element) bool ) SortedList {
    return SortedList{less: less}
}

func (l *SortedList) Add(x int) {
    p := &(l.Head)
    v := &Element{Value:x}
    for i:= 0; i < l.Len; i++ {
        if l.less(v, (*p).Next) { break }
        p = (*p).Next
    }
    (*v).Next = (*p).Next
    p.Next =  v
    l.Len++
}
```

программа-пример sortedlist2.go:

```
package main

import (
    "fmt"
    "order/sorted"
)
```

```
func main() {
    l := sorted.NewSortedList(func(p, q *sorted.Element) bool {
        return (*p).Value < (*q).Value
    })
    l.Add(2)
    l.Add(4)
    l.Add(1)
    l.Add(8)
    l.Add(5)
    p := l.Head
    for i := 0; i < l.Len; i++ {
        p = *(p.Next)
        fmt.Printf("%5d", p.Value)
    }
}
```

Просто радость.

Но для полного счастья надо бы разрешить делать `Value` произвольным, т.е. велиной типа `interface{}`. Так и сделаем. Заодно уберём из доступа поля `Len`, `Next` и `Head` - нефиг юзеру их менять, не приведи господи. Конечно, при этом надо реализовать методы `Do` и `Len`, возможно ещё какие-то. Говорить тут долго ни к чему, просто посмотрим соответствующий код:
package sorted:

```
package sorted

type Element struct {
    next  *Element
    Value interface{}
}

type SortedList struct {
    head Element // sentinel list Element;
    // it's located at the top of the list
    len  int // current list length excluding sentinel
    less func(*Element, *Element) bool
}

func NewSortedList(less func(*Element, *Element) bool) SortedList {
    return SortedList{less: less}
```

```
}

func (l *SortedList) Add(x interface{}) {
    p := &(l.head)
    v := &Element{Value: x}
    for i := 0; i < l.len; i++ {
        if l.less(v, (*p).next) {
            break
        }
        p = (*p).next
    }
    (*v).next = (*p).next
    p.next = v
    l.len++
}

func (l SortedList) Do(f func(v interface{})) {
    p := l.head
    for i := 0; i < l.len; i++ {
        p = *(p.next)
        f(p.Value)
    }
}

func (l SortedList) Len() int {
    return l.len
}
```

программа sortedlist3.go:

```
package main

import (
    "fmt"
    "order/sorted"
)

func main() {
    l := sorted.NewSortedList(func(p, q *sorted.Element) bool {
        return (*p).Value.(int) < (*q).Value.(int)
    })
```

```
    l.Add(2)
    l.Add(4)
    l.Add(1)
    l.Add(8)
    l.Add(5)

    l.Do(func(v interface{}) {
        fmt.Printf("%5d", v.(int))
    })

}
```

И, вдохновляясь последним примером, просто хочется выполнить самостоятельно вот такое **задание**:
Реализовать двусвязный сортированный список, можно даже запихать его в тот же пакет `sorted`, хоть это и странно, но для учебных целей норм. При этом самим список не строить, а воспользоваться двусвязным списком из `container/list`, обогатив его функцией сравнения элементов. Да, и тогда все лишнее из `container/list`, всё то, что мы не сочтём нужным предоставлять пользователю нашего сортированного списка (а список постоянно сортированный, так что надо бы закрыть пользователю возможность добавлять элементы просто так, мимо соответствующего метода, а также изменять элементы, т.е. их значения, конечно). Ну, так просто не будем вставлять эти методы в наш пакет. А вот те, что мы сочтём нужным оставить - просто напишем их реализации именно для нашего двусвязного сортированного списка, благо писать реализации тут совсем легко - просто вызвать соответствующие методы из `container/list`